1989

Shared Memory vs. Message Passing Architectures: An Application Based Study

Margaret Martonosi and Anoop Gupta
(Draft: Nov 9. 1988. 7:30 pm)
Computer Systems Laboratory
Stanford University. CA 94305

Abstract

The diminishing differences between the hardware structure of shared memory and message passing parallel computers mandate a new evaluation of the tradeoffs these architectures impose on the development and performance of applications. In a message passing computer. some message traffic is used to perform interprocessor updates which maintain consistency between the various processors' data. We consider this traffic to be analogous to global bus traffic needed in a shared memory computer for hardware cache consistency. Using LocusRoute. a global router for standard cells. as a case study. we investigate the level of traffic required to maintain consistency of data with each of the two architectures. By explicitly varying the frequency of interprocessor updates. the level of traffic in the message passing approach can be reduced to as little as 1% of the traffic in the shared memory approach while still obtaining solution quality within 10% of the quality given by the shared memory version. We show that exploiting locality. in the way wires to be routed are assigned to processors. can further lower this message traffic by as much as 67%. However. the degree to which locality can be exploited may be limited by the opposing requirement that the application be load balanced. as well as by limited locality in the data set.

1 Introduction

In recent years. there has been much debate about the relative merits of shared memory and message passing parallel architectures. The previously large distinctions between the two approaches. however. are now diminishing. The main drawbacks of the early message passing computers. such as the NCUBE/ten [11] were the high network latency and the large message reception overhead. These characteristics forced programmers to exploit only large grain parallelism. With the development of new message passing computers. such as the Ametek Series 2010 and the Message Driven Processor [3.9]. things are rapidly changing. Using specialized routing chips and the technique of wormhole routing [6]. the network latencies have been reduced by 2-3 orders of magnitude. Similarly. using dedicated hardware to copy messages to and from the network and using innovative memory mapping techniques [10.18]. the message reception overhead has been cut down by 1-2 orders of magnitude. These reductions enable current machines to exploit parallelism at a fairly fine grain. Furthermore. it is now possible to approximate aspects of the shared-memory model. since sending a message to a remote processor requesting an update of some global data is no longer an unreasonably long operation.

On the other hand. we see that efforts to scale shared memory machines led them to resemble message passing computers in several ways. Traditional shared memory architectures used a shared global bus to memory [5] and could only accommodate a limited number of processors before the global bus connecting processors and memory became saturated. Because of the limited scalability of these architectures. designers of shared memory machines are now turning to architectures using processor clusters with directory-based cache coherence schemes between the clusters [19.1]. hierarchical architectures with more than one level of shared buses such as the Encore UltraMax [13.4]. or architectures with multistage processor interconnection networks

Submitted for Publication

like the BBN Butterfly [12] and IBM RP3 [16]. In directory based approaches, consistency operations between clusters are performed on a point-to-point basis, with invalidations going only to the clusters that need them. Also, the latency of non-local communication can be as much as an order of magnitude larger than that of a local access in all of these machines. These two characteristics, point to point communication and high non-local reference latency, increase the cost of non-local traffic in a shared memory approach, and force the programmer of these shared memory machines to consider more carefully the effect of locality.

While the gap between the two architectures is narrowing, there are still fundamental differences between the two which force tradeoffs between the two architectures. The shared memory architecture considered in this paper has a single global address space with hardware to guarantee the consistency of data in the processor caches. In this case, cache coherence protocols enforce consistency of data [2]. In the message passing architectures, each processor has a separate address space, and exchanges of information occur by sending messages on the network. One type of message passed on the network is for updating distributed data held by all the processors to periodically make it consistent with the other processors' data. This message traffic for updates in the message passing case is analogous to the cache coherence traffic used in the shared memory case. In the message passing case, this traffic is explicitly controlled by the application programmer, who decides when and how to perform updates between processors. In the shared memory case, the traffic is implicitly controlled by the cache consistency hardware, which relieves the programmer from the process of maintaining data consistency.

In many cases, however, the level of consistency enforced by the shared memory computer may be more than is needed by a particular application. In such cases the message passing computer may be superior, because it allows the application programmer to control the degree of consistency explicitly. In this paper, we explore several such tradeoffs between shared-memory and message passing architectures using the LocusRoute [17] application as a case study. (LocusRoute is a commercial quality routing program for standard cells and is now being widely used at Stanford for parallel processing studies.) Our results comparing network traffic show that the message passing version of the program generates only 1% of the traffic that the shared memory version does, while the degradation in the quality of the routing is less than 10%. Another issue this paper addresses is the sensitivity of the network traffic to the exploitation of locality in the data set. In our study, we exploit locality by assigning wires which are physically close to each other in the circuit, to the same processor. Specifically, we show that message passing architectures can reduce their network traffic by more than 50% by exploiting locality, while also improving solution quality. The effect on shared memory architectures, while not so large, is also significant. Some other issues regarding exploiting of locality and execution time are also discussed.

The rest of the paper is structured as follows. The next section gives information about the LocusRoute application and the simulation tools used to collect data for the architectural comparison. Section 3 describes the changes made to the original shared memory LocusRoute to convert it to a message passing style. Section 4 presents our results on network traffic for the two architectures under varying assumptions, and Section 5 shows the reduction of network traffic made possible by exploiting locality. Section 6 presents conclusions based on this data, and suggests further areas to explore.

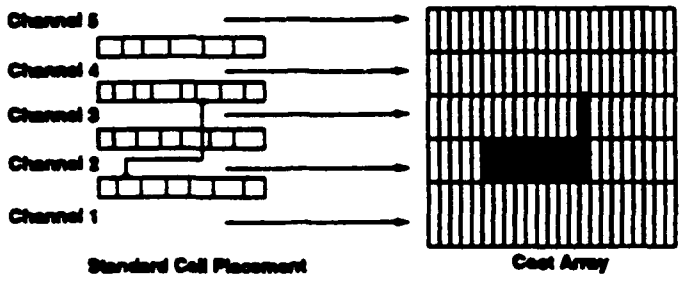

Figure 1: Standard cell placement and corresponding cost array.

2 Applications, Tools and Methodology

To understand the architectural comparisons being made, one must understand the application the data is based on, and the tools used to make the measurements. Our starting point was the version of LocusRoute written for a shared memory machine. This code was converted, as described in Section 3, to a message passing style. Because there was no message passing computer available to run the code, we used CBS, a simulator for parallel message passing machines. With CBS, detailed statistics on execution time and network behavior are readily available. To make network traffic comparisons between the message passing version and the shared memory version, another program, also described below, was written to estimate the amount of bus traffic required by the shared memory approach.

2.1 LocusRoute

LocusRoute [17] is an industrial quality router for VLSI standard cells developed by Jonathan Rose at Stanford University. LocusRoute routes the wires of a given standard cell placement, while attempting to minimize the overall circuit area. To do this, it maintains a global data structure known as the `Cost Array`. The vertical dimension of the array is the number of routing channels in the circuit, and the horizontal dimension of the array is the number of routing grids. The `Cost Array` keeps a record of the number of wires running through each sector of the circuit. Each wire is routed along the path with the minimal sum of the cost array entries. Figure 1 shows a standard cell circuit and one of its wires, with the corresponding `Cost Array`. The highlighted portions of the cost array will be incremented if this route is chosen.

In addition to producing the routed circuit, LocusRoute also computes a measure of the solution's quality. Quality, also referred to in this paper as the *circuit height*, is computed as follows. For each channel, the number of wires using the channel will vary across the width of the circuit. The number of routing tracks required by the channel is the maximum number of wires running through the channel at any point. The *circuit height* is the total number of routing tracks required for all channels.

The `Cost Array` is the central data structure for the LocusRoute application, and it accounts for almost all of the shared data references made by LocusRoute. Therefore, studying the reference patterns to the cost array will provide an excellent approximation to the application's memory reference behavior as a whole. Examining the references to the cost array for one wire, we see that LocusRoute starts with a series of reads to explore possible routes for the wire.

LocusRoute reads every location of the cost array along the paths being considered. This is followed by a smaller stream of writes, as the cost array is updated along the final path of the wire. This basic sequence of reads and writes occurs for each wire, with several processors routing wires in parallel. Performing several iterations of routing improves the final solution quality, but, before rerouting a wire for an iteration after the first one, the processor must "rip up" the old routing of the wire by decrementing the cost array locations in its path. These rip up operations are the second type of writes performed on the cost array.

Consistency of the cost array is an important issue in this paper, and one with serious implications on the amount of traffic, so it is important to understand how, and to what extent, consistency is maintained in the shared memory version of LocusRoute. To avoid the performance bottleneck a lock would impose, accesses to the cost array are not locked. This implies that simultaneous operations on the same element of the cost array may result in one of the operations being lost. As previously stated, LocusRoute is a optimization problem, and can tolerate a certain amount of inconsistency. With the number of processors the shared memory version currently uses (up to 16), the probability of simulataneous writes is very low, and experiments indicate that the quality is not degraded. Except for this, consistency in the cost array is maintained at the hardware level, by the cache coherency hardware.

When running the experiments, two benchmark circuits were used. The first circuit, bnrE has 420 wires, a size of 10 channels by 341 routing grids, and represents an actual standard cell circuit developed at Bell-Northern Research Ltd. The second circuit, MDC, has 573 wires with a size of 12 channels by 386 routing grids, and was designed at the University of Toronto Microelectronic Development Centre.

2.2 CBS: A Message Passing Architecture Simulator

Execution of LocusRoute on a message passing computer was simulated using a program called CBS. CBS [15] is a C++ program written by Andreas Nowatzyk at Carnegie Mellon University which simulates the behavior of a k-ary n-dimensional hypercube machine (with a total of k^n processors). For the experiments described here, CBS simulated a machine with deterministic wormhole routing using the E-cube routing algorithm [14,21], and with the dimension n, always equal to two (mesh interconnection). The use of wormhole routing minimizes the effect of the distance between destination and source, making the assignment of processes to processing elements less critical. Research by Dally [7,8] indicates that low-dimensional networks have greater channel bandwidth, and better hot-spot throughput than do high-dimensional networks. These two features give the simulated machine low-latency, high-bandwidth communication performance which makes it competitive with the shared memory machine.

CBS uses a detailed simulation model to produce its network statistics. CBS simulates the behavior of the processor interconnection network at the level of individual flow control units (in this case, single bytes) flowing between processors. There are unidirectional channels connecting a processor to its North and East neighbors. This means that a packet must travel all the way around the network to talk to its West neighbor. The network performance is specified with two parameters: *t_delay* and *c_delay*. T_delay is the time required for 1 byte to travel one hop on the network, and c_delay, the time required for the entire packet to be copied down from the processor node to the message network, or up from the message network to the destination processor node. Assuming no delays due to contention, the total time required for a packet of length L to travel D hops on the network is $2c\_delay + t\_delay(D + L)$. To simulate the execution time of the node processors, a *delay* statement is provided which blocks the running processor for the number of time units specified. Timings obtained from the Encore microsecond clock

were used as arguments for the delay statement.

For the purpose of concreteness, we chose to set the performance parameters to model the behavior of the Ametek Series 2010 Message Passing Multicomputer [18.10]. A packet of length L travelling D hops on the Ametek requires $CopyTime + HopTime(2D + L)$. CopyTime is the time required to copy the message from the network to the node processor's address space, which depends on the message length. This can be performed at about 50MB/s. Assuming an average message length of 200 bytes, we chose CopyTime to equal 4000 ns, so c_delay was set to half of that, or 2000 ns. [1] HopTime for the Ametek is defined as the time it takes for one byte of a packet to advance one hop, assuming that the route has already been established. This is stated to be 50 ns. (Establishing the route is slower, so the head of a packet requires two HopTimes to advance one hop.) To make the Ametek packet latency equation conform to the CBS form, we factored out the 2 to get: $CopyTime + 2HopTime(D + L/2)$. Now we can set t_delay equal to $2HopTime$ or 100 ns. When the simulation is run, the number of bytes in a packet is always cut in half, so that the L/2 term in the Ametek equation matches the L term in the CBS equation. Also, since the Ametek 2010's processing elements are about five times faster than the Multimax's processing elements, we divide the times obtained on the Encore processing elements by a factor of five before using them as arguments to the delay statement.

2.3 Shared Memory Traffic Evaluation

While for the message passing implementation, the network traffic is directly given by CBS, there is no simple way of estimating traffic for the shared memory implementation. In order to make comparisons of the traffic required for message passing and shared memory approaches, we need a method for measuring the traffic generated by LocusRoute on a shared memory machine. Although we have the capability to directly trace all memory references [20.22], these direct methods require a large amount of memory, which limits the portion of the program that can be traced to about 1 wire per processor. As will be described in Section 3, updates in the message passing approach can occur at time intervals greater than the total time being monitored by these detailed trace methods, making traffic comparison difficult. Instead we have chosen to modify the shared memory version of LocusRoute to record information about the memory reference stream over the whole execution time and use this information to estimate total traffic.

Before explaining the traffic estimating program, we will first explain in detail the type of machine to be simulated. This work considers a shared memory multiprocessor with a single global bus that all processors use to access memory. Each processor has a private cache memory, and consistency is maintained by dedicated cache coherence hardware using a Write Back with Invalidate scheme. The traffic being measured in the shared memory version is the traffic on the shared global bus. [2]

With the above machine model in mind, we now describe a method for estimating the bus traffic. The uniprocessor version of LocusRoute is modified to record memory reference information. Data is printed to a trace file whenever one of four types of events occurs. The four event types are described below:

1. **Wire event:** Whenever routing of a new wire is begun, the time and wire name are recorded as a *wire event* in the trace file.

[1] C_delay is a parameter set at the time the simulator is compiled. Therefore, one cannot compute c_delay dynamically.

[2] Our traffic estimating program can simulate non-bus-oriented architectures as well, but since we are only considering 16 processing elements, we present data only for the bus-based scheme.

2. **Iteration event:** When a new iteration is begun, the time at which it was begun is recorded as an *iteration event* in the trace file. These first two events are needed by the traffic evaluator for interleaving the execution among several processors.

3. **Read event:** LocusRoute is structured so that a processor reads data from the cost array when it evaluates possible routes for a wire segment. For each route considered, a processor reads all the locations of the cost array along the path of the route. At the beginning of each of these read sequences, a *read event* is recorded in the trace file, with the time, and the locations affected by the reads.

4. **Write event:** A processor writes data at two times: (i) at the end of exploring alternative routes, and (ii) when it does a rip-up before starting to explore routes. Both of these are recorded as write events in the trace file along with the time at which they are begun, and the locations affected by the writes.

All events are assumed to be atomic: one time is recorded in the trace file at the beginning of the read or write sequence, and all reads or writes associated with that event are assumed to occur at that time.

The trace file described above records the relevant memory access information about a *uniprocessor* run of LocusRoute. The events recorded in the trace file give enough information to interleave execution among more processors, so that *multiprocessor* traffic data can be estimated using the steps described here. First, the simulator decides on an assignment of wires to processors, using one of the heuristics from Section 3. At this point, each event in the trace is associated with a certain wire, and each wire has now been assigned to a processor, so all the events are now associated with the processor executing them. Events for each processor can be interleaved using the times recorded in the trace file. To simulate the traffic, events are pulled off the interleaved event queue and handled in order. The cache coherence protocol implemented is Write Back with Invalidate Scheme [2]. The first write to any cache line results in a bus operation which causes all other caches to invalidate that line if it is present. Subsequent reads or writes by that processor do not result in any bus traffic. A read or write by another processor to that cache line causes that processor to become the owner, and forces the previous owner to invalidate the line from its cache.

Like any such measurement system, ours has its inaccuracies, which we will list here. First, the system simulates infinite caches. Lines are only written back to memory for coherency reasons, never simply for replacement. Because the data structure we are studying is smaller than most multiprocessor caches (about 8000 bytes), this is not a major flaw. Second, we assume that read and write *events*, which are conglomerations of several read or writes, occur atomically. In actual execution, these operations occur as sequences of operations in tight (single instruction) loops. The atomic assumption only leads to inaccuracy when a read event and a write event, or two write events, that should be occurring simultaneously become serialized by the simulator. If these events were occurring in the same area of the cost array, then their simultaneous execution would lead to multiple invalidations and refetches. If they are serialized, at most one invalidation and refetch will occur. This will cause the simulator to slightly underestimate the total traffic. This is not a major effect, because write operations are relatively infrequent, so simultaneous reads and writes to the same cache line are highly improbable. Both of the inaccuracies tend to slightly underestimate the total traffic so that the numbers given in Section 5 may be considered lower bounds on the actual traffic.

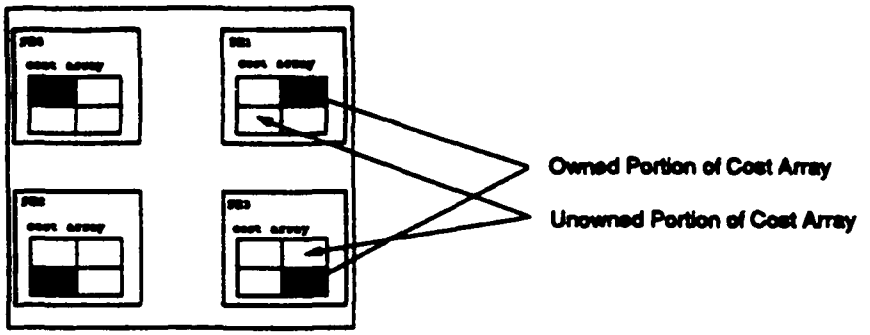

Figure 2: Division of the cost array among processors.

3 Implementing LocusRoute for a Message Passing Machine

Finally, before we go on to discuss the results, we need to specify how we mapped LocusRoute to a message passing architecture. Because the message passing machine has distributed memory, implementing LocusRoute required changes in the distribution and updating of information between processors. To reduce message traffic on the network, a static method of assigning wires to processors was used, rather than allowing processors to send requests out when they need a wire. These changes are described below.

3.1 Distribution of Data Structures

The most important data structure in the program, as previously stated, is the `Cost Array`. In the shared memory version, all processors have access to a single copy of the cost array. The message passing architecture forces the programmer to decide how the cost array should be handled in a machine with distributed memory. We chose to divide the cost array into sections, with each processor being the owner of one section. Each processor is, however, allowed to have a view of the whole cost array. The portion that is owned by that processor will be as consistent as possible, while the other portions of the cost array are less consistent, but still usable. Consider, for example, a four processor case. Figure 2 shows each processor's cost array, with the portion that it owns highlighted. Although, the unowned portions of each processor's cost array may not be accurate, the processor is still allowed to make use of them. Thus, if there are 4 processors, the bottom left processing element will own the bottom left fourth of the cost array, but it will also have a copy of the rest of the cost array which it can use. In all future discussion, the processor which owns a certain region of the cost array will be called the *owner processor* for that region, and the region itself will be called the *owned region*.

3.2 Maintaining Cost Array Consistency

No circuit is perfectly local, that is, wires assigned to one processing element will extend into regions owned by other processing elements. Consequently, using the scheme discussed above cost array updates between processors are needed. Many different methods of performing these updates are possible: one can experiment with the frequency of updates, as well as how the updates are initiated. The update frequency was allowed to vary, with updates occurring at time intervals on the order of the time it takes to route one wire. The decision of which update

strategy to use depends heavily on the underlying computer architecture. We have considered two main types of updates, and variations on these. The two types of updates considered are *sender initiated* updates, and *receiver initiated* updates, as well as a mixture of these two.

With sender initiated updates, the processor which determines that an update is necessary is the one to send out the data. With receiver initiated updates, the processor which determines that an update is necessary sends a request packet to another processor, and the destination processor then sends back the requested update data. The architectural dependencies in these schemes should be clear. For receiver initiated updates to be useful, the latency of the network as well as the message reception time, must be low, so that the requesting processor spends a minimal amount of time idle, waiting for the requested data. On the other hand, our results shown in Section 4 indicate that sender initiated updates tend to send out more bytes than receiver initiated, and therefore place a greater premium on high network bandwidth.

3.3 Wire Assignment

One advantage of the shared memory architecture is that the wires to be routed can be easily allocated to processors dynamically, using a distributed loop. [3] In the message passing architecture, dynamic wire allocation requires message transactions on the network. In our implementation, processors only retrieve messages queued for them at the end of each wire routed. Assuming that the processor receiving the task requests is also routing wires, the potential wait for a wire task is large, because in this case, a processor may have to wait for an entire wire to be routed before the wire assignment processor even receives the message. With this in mind, a static method of wire assignment was used. Because of the division of cost array into owned regions, the algorithm benefits from a method of wire assignment that attempts to assign wires to the processor that owns the region they run through. We use a very simple heuristic developed to achieve this goal. We assign wires to the owner processor of the lower leftmost pin of the wire.

To control the amount of locality exploited, a parameter *ThresholdCost* is provided. If a wire's "cost", a function of its length, is less than the threshold, it will be assigned using the heuristic described above. Otherwise, it is held in a pool of unassigned wires, and is assigned to a processor at the end of the wire assignment phase. The processor it is assigned to is the processor whose total wire cost is the current minimum. With this method, a high value of ThresholdCost results in a wire assignment that is based primarily on locality, while a low value of ThresholdCost results in a wire assignment that is based mostly on load balancing.

4 Traffic in Shared Memory and Message Passing Architectures

In this section, we present results on the traffic generated by shared memory and message passing architectures. We think this is a useful measure because in both architectures overly high network traffic results in a performance penalty. For example, since the message passing architecture forces the cost array to be distributed across the processors, periodic update messages are needed to keep the processors' views of the cost array consistent. There is computational overhead associated with sending and receiving these messages, so one would like to update as infrequently

[3] Associated with a distributed loop is a locked index variable. To get the next wire to be routed, a processor obtains the lock, reads and increments the index of the next wire to be routed, and releases the lock.

as possible. Similarly, in the shared memory architecture, hardware cache consistency protocols cause extra global bus traffic due to cache line invalidations and any subsequent refetches that may be needed. These operations cause the processor to stall, and also represent a performance overhead. Although the degree to which network traffic translates to performance overhead will be different for the two architectures, we contend that a comparison of the network traffic between the two architectures is itself useful.

This section will show that traffic for the shared memory architecture is a strong function of the cache line size, while traffic in the message passing architecture is explicitly controlled by the programmer. This explicit control allows the traffic to be reduced by more than two orders of magnitude compared to the shared memory traffic, and the program still gives comparable solution quality.

4.1 Traffic in the Shared Memory Architecture

Here we consider traffic in the shared memory approach. Traffic in the shared memory approach is made up of 3 parts. First, the processor's very first access to a location always results in a miss, and brings the line into the cache. Second, the first write to a clean location causes a word write on the shared bus. The other processors see this write and invalidate that cache line if it is in their cache. Third, once a line has been invalidated by a cache, it may need the line again. This leads to refetches of data from memory. Traffic in the shared memory architecture is, clearly, a function of the cache coherence protocol used, and the line size of the cache. For all the results given here, the coherence protocol used was a Write Back with Invalidate scheme [2]. The line size of the cache was allowed to vary.

Increases in the cache line size can have the effect of either increasing or decreasing traffic. There are two factors which will increase traffic with increasing line size. First, with an increased line size, data items that will never be used are more likely to be brought into the cache. This will increase the traffic on the bus. Second, increasing the line size means there will be more data in the cache (under the infinite cache assumption) and this means that processors are more likely to interfere with each other. With more data in the caches, processors are more likely to force invalidations in other caches. These invalidations, as well as the subsequent refetches, also cause the traffic to increase. On the other hand, it is possible for a longer cache line to cause a traffic decrease as well. If there are several shared data items stored relatively close to each other, then a single invalidation of a long cache line could cause them to all be invalidated in one operation. This can save bytes over the case of several individual invalidations, and cause the traffic to decrease. Because this last situation happens infrequently, its effect is minor compared to the first two. Thus, we expect that increasing the cache line size will lead to an increase in the number of bytes transferred.

Table 1 shows the shared memory bus traffic as a function of the cache line size. As predicted, the data clearly shows that the traffic increases significantly as the line size increases. For example, in the MDC circuit, a cache line size of 4 bytes causes the total traffic to be 932,976 bytes while a larger cache line size of 32 bytes causes the traffic to increase sharply to 5,840,280, more than five times as much.

Solution quality and execution time are not available for the wire assignments shown in Table 1 because the actual shared memory implementation uses a dynamic wire assignment. The simulator does not actually route wires, so it cannot output solution quality or execution time values. However, for comparison with the message passing figures presented later in the paper, the shared memory version of LocusRoute running on an Encore Multimax with 16

Table 1: Traffic as a function of cache line size in shared memory version.

Circuit	Cache Line Size	Bytes Transferred
bnrE	4	769.788
	8	1.306.152
	16	2.429.764
	32	4.712.540
MDC	4	932.976
	8	1.596.940
	16	3.043.941
	32	5.840.280

processors can route the bnrE circuit in a time of 5.59 seconds with a height of 136. For MDC, the solution quality is 144 in a time of 5.78 seconds.

4.2 Traffic in the Message Passing Architecture

Now that we have presented data from the shared memory architecture, we move on to the data from the message passing architecture, comparing the two as we go. Traffic in the message passing approach is determined by the programmer, subject to the constraint of acceptable solution quality. The programmer controls the size of messages, as well as their frequency. The results given in Table 2 show the traffic required by the message passing approach with varying update strategies. All the results given are for 16 processors. Because of the explicit tradeoffs in the message passing approach between network traffic, solution quality, and execution time, one cannot discuss the amount of network traffic required without also discussing the update strategies used and the resulting solution quality and execution time. Table 2 shows data for three different update strategies (discussed in Section 3.2). Recall that, in a sender initiated strategy, it is the sender of the update that determines when the update should be sent. In the receiver initiated strategy, the processor wishing to receive an update sends a request to another processor, who then returns the data. The third strategy is a mixture of the other two. In this third mixed approach, sender initiated updates occur with the same frequency as in the purely sender initiated strategy, and receiver initiated updates occur with the same frequency as in the purely receiver initiated approach.

4.2.1 Network Traffic

There are two points to be made about the message passing network traffic. First, there is a large difference between bytes transferred for sender initiated and receiver initiated. Intuitively, one would expect the receiver initiated approach to be more efficient in terms of network traffic, because data is only sent when it is specifically requested. In contrast, in the sender initiated approach, data is sent periodically, regardless of whether the destination processor is routing wires in that area and needs that data or not. Consequently, one would expect a larger number of bytes to be necessary to get the same quality as receiver initiated. The data, shown in Table 2 bears out this intuition. Receiver initiated transfers use anywhere from 44% to 71% fewer bytes than sender initiated to produce similar quality results. Because sending and receiving

Table 2: Traffic in the message passing version.

Circuit	Update method	Circuit Height	Execution Time	Bytes Transferred
bnrE	Sender Initiated	145	1.603	156468
bnrE	Receiver Initiated	150	1.210	87572
bnrE	Mixed	146	1.519	245270
MDC	Sender Initiated	150	2.171	236304
MDC	Receiver Initiated	156	1.635	85646
MDC	Mixed	153	2.208	324914

messages has a computational overhead. the savings in network traffic translate to time savings as well. For all the trials given in Table 2. the receiver initiated method is about 20% faster than the corresponding run using the sender initiated method.

The second point to note about the network traffic on the message passing architecture is that. even with the less efficient sender initiated method. the message passing network traffic is more than an order of magnitude less than that for shared memory. The huge difference may. at first glance. be surprising. but it can be explained by two factors. First. the updates being performed in the message passing version occur. at most. once per wire. In the data shown in Table 2. they occur approximately every two wires. Second. these updates can be thought of as constituting a very loose form of coherence protocol. There are several differences between this protocol and the strict one implemented on the shared memory system. Because updates occur no more frequently than once per wire. the write performed at the wire rip up stage is handled at the same time as the write performed at the wire routing stage. Because much of the wire's path will remain the same after rerouting. these two writes will often cancel each other. and many of the locations will not need to be updated at all. [4] This removes many of the write operations. a significant accomplishment since writes are the cause for over 80% of the bytes transferred in the shared memory version.

4.2.2 Solution Quality

A discussion of network traffic in the message passing approach is incomplete without also discussing the solution quality and execution time required. The first point we note is that the solution quality. that is the height of the circuit. has degraded slightly from the quality given by the shared memory version. but is still acceptable. Recall that the solution quality for the shared memory approach was 136 for bnrE and 144 for MDC.

Some quality degradation can be expected in the message passing approach. because less information is available to each processor as it is routing. For example. the cache coherence hardware on the shared memory machine guarantees a perfectly consistent view for all processors at all times. The only inconsistency in the shared memory approach comes from not locking the cost array. and as previously explained. this has no noticeable effect on the quality. By contrast. in the message passing approach. updates occur only after the processors have each routed

[4] In the message passing implementation, a *delta array* is maintained which records changes made to the cost array. If no changes are made to a location. or the changes cancel each other. updates for that location need not be sent.

one or more wires. This leads to a much larger level of inconsistency in the processors' cost arrays. In the worst case, the solution quality in the message passing approach has degraded by 10% from the shared memory solution quality. LocusRoute is intended to be used with a standard cell placement program, so that once the placement program has decided on a placement, LocusRoute performs the routing for that placement. LocusRoute returns the circuit height as a measure of that particular *placement's* quality. Thus, in this situation, slight declines in the quality of the routing can be tolerated. However, if LocusRoute is used to produce the final routing for a circuit that is to be mass produced, this increased area could become quite significant. For these cases, this characteristic of the message passing implementation of LocusRoute could make it undesirable.

Section 5 will discuss how the quality of the solution can be improved somewhat by exploiting locality in the wires. However, another obvious way to improve the solution quality is by increasing the frequency of updates. The best combination of execution time and solution quality obtained for bnrE was a solution quality of 136 with an execution time of 1.7 seconds and 843,542 bytes transferred. Note that the quality of this run equals the quality given by the shared memory version. The bytes transferred, while much larger than any of those in Table 2 is still about a factor of two less than those measured for the shared memory version. Results such as this indicate the robustness of the LocusRoute algorithm to inconsistencies in the cost array. For LocusRoute, and other applications like it, hardware cache consistency seems to impose a large cost on the execution of the program, without giving compensating benefits.

4.2.3 Execution Time

Now, we turn to the execution time. Execution time for the message passing version of LocusRoute depends heavily on the wire assignment strategy for two reasons. If the wires are not assigned in a way that will balance the load on the processors, execution time will suffer greatly. This is discussed in Section 5.3.2. On the other hand, if the wire assignment does not exploit locality well, more update packets will need to be sent to get the same quality, and the extra processing time spent sending and receiving messages will show up in the final execution time of the run. In general, the execution times given in Table 2 are much faster than those for the shared memory runs. However, recall that CBS is simulating processors which are five times faster than the processors used when timing the shared memory version. A rough comparison can be obtained by multiplying the execution times of the message passing version by five. The fastest execution time obtained overall is 0.97 seconds for the receiver initiated scheme. Multiplying by five, the execution time becomes 4.85 seconds which is still 13% faster than the execution time from the shared memory case. The quality obtained in the message passing run being considered was only 7% worse than the shared memory quality. Note that simple multiplication by a factor of five when comparing the execution times favors the shared memory architecture. This is because if the processors in the shared memory machine really were five times faster, there would be more contention on the bus, and the overall performance would not improve by a factor of five.

5 Effect of Locality

The previous section compared the network traffic required by LocusRoute in the shared memory and message passing architectures. Here we examine the effectiveness of exploiting locality to reduce this traffic in both architectures. Locality, here, is a measure of how often a processor is

routing wires within its owned region or regions close by. (A quantitative measure is described in Section 5.3.1.) Both architectures benefit differently from exploiting locality. Message passing architectures benefit from locality because the need for message traffic to produce a certain level of solution quality is reduced. This is because improving the locality of the wires routed by each processor means that each processor will have a better view of the part of the cost array it is routing in, and fewer updates will be needed. Shared memory architectures benefit from locality through better cache behavior. Specifically, shared memory architectures benefit because of two factors—better spatial locality, and less interference between processors causing cache coherence traffic.

In the past, locality has not played a major part in the design of shared memory parallel programs. In a traditional global bus shared memory computer, all memory is equally accessible to all processors. However, as hierarchical approaches are used to scale shared memory multicomputers, this is no longer true. The current trend towards hierarchical shared memory machines, in which a local reference can be more than an order of magnitude faster than a non-local reference implies that locality must become an important part of future program design. In this section we present data that indicates that implementations taking advantage of locality can reduce the total network traffic by as much as 67%.

5.1 Effect of Locality in the Shared Memory Architecture

In this subsection, we study the effects of locality on the interconnection network traffic generated by a shared-memory architecture. Table 3 shows the amount of network traffic generated as a function of differing amounts of locality exploited in the application. The extreme non-local case is taken to be the round robin wire assignment to processors, and the extreme local case (ThresholdCost = infinity) is taken to be the one where each wire is assigned to the processor whose owned region contains its lower left pin. We also consider two cases with intermediate locality.

Table 3: Effect of locality in shared memory version.

Circuit	Allocation strategy	Total Wires	Wires Held for round robin asmt.	Bytes Transferred	Reduction from round robin (%)
bnrE	round robin	420	420	1,306,152	—
	ThresholdCost = 30		209	1,299,580	0.5
	ThresholdCost = 1000		25	1,275,492	2.3
	ThresholdCost = inf		0	1,219,576	6.6
MDC	round robin	573	573	1,596,940	—
	ThresholdCost = 30		263	1,608,520	-0.7
	ThresholdCost = 1000		38	1,593,104	0.2
	ThresholdCost = inf		0	1,516,468	5.0

For bnrE with 8 byte cache lines, total global bus traffic can be reduced 6.6% by taking advantage of locality in the assignment of wires. It is clear that locality is not producing the significant benefit we had expected. The reason for this is that cache lines contain too many cost array entries. Since each cost array entry is one byte, a cache line holds eight cost array

entries. This means that for each byte accessed, seven of its neighbors are brought in as well. The increase in traffic brought about by interference between the caches maintaining coherence is almost as big as the decrease in traffic due to locality. To check this theory, we increase the size of the cost array entries to 4 byte integers, so an 8 byte cache line will hold only two of them. In this case, the total traffic for a round robin wire assignment is higher— up to 1,799,984 bytes, but the percent gain from exploiting locality is higher as well. For 4 byte array entries with an 8 byte cache line, changing to the infinite ThresholdCost wire assignment reduces the traffic by 15.6%. The reason for this bigger reduction is the following. With a cache line that holds more array entries, the probability of interference between processors causing invalidations is higher. Intuitively with these 1 byte cache entries, it is harder for a processor to be active in only a small section of the cost array, because every time an array entry is accessed, eight entries are fetched into the cache. If each processor has many "extra" entries in its cache, the cache coherence traffic will reduce the benefit possible from exploiting locality. Obviously, we are not concluding that all variables be made as large as possible, so that they interfere less with each other. Rather, the conclusion is that care must be used when allocating memory for write-shared data items. The notion, stemming from uniprocessor caches, that dense data will display better locality, and therefore better caching behavior, is no longer true when speaking of multiprocessor cache coherent systems. In multiprocessor cache coherent systems, the benefits of data density must be traded off with the penalty of increased coherency traffic.

5.2 Effect of Locality in the Message Passing Approach

Having examined the traffic in the shared memory case, we move to the message passing case, where the effect of locality is much more significant. Data in Table 4 shows the effect of various wire assignment strategies on the quality of the routed circuit, the execution time, and the number of bytes transferred.

One can see that in general, wire assignments which do not take advantage of locality, such as round robin, result in poorer solution quality than those that do, such as assignments made with infinite ThresholdCost. The average quality improvement due to locality over the cases shown in Table 4 is about 4%. While small, this improvement is quite significant, because the results given are all quite close to optimal anyway, so even a small percentage improvement is difficult to achieve. This data indicates that it is best to have a single processor route the wires in one area, because that processor will have more accurate information about the cost array in that area.

Having seen that exploiting locality improves the solution quality, the next question is, what is the effect of locality on the number of bytes transferred? This depends heavily on the type of update strategy used. In the sender initiated scheme, updates are sent out for any owned region in the sender's array that has changed, so the only reduction in traffic will be due to changes being made in fewer and smaller regions of the cost array. The change in bytes transferred for sender initiated updates from a round robin assignment to a local assignment with infinite ThresholdCost is 11.4%. The receiver initiated scheme will be more sensitive to locality, because in this strategy, low locality results in frequent interprocessor data requests. A processor only requests an update if it has routed in a certain owned region a specific number of times. If, by exploiting locality, we reduce the frequency with which update requests need to be made, we can dramatically reduce the message traffic. The data bears out this prediction, with a traffic reduction of more than a factor of two in both circuits, when going from a round robin assignment policy to a local one. Because the mixed strategy is made up partly of receiver initiated requests, which are highly sensitive to locality and partly of sender initiated requests,

Table 4: Effect of locality in the message passing version.

Circuit	Update method	Wire Assignment	Circuit Height	Exec. Time	Bytes Transferred	Reduc. from rnd-rbn (%)
bnrE	Sender Initiated	round robin	145	1.603	156468	—
		Thr_Cost = 30	138	1.467	149562	4.4
		Thr_Cost = 1000	139	1.647	141206	9.8
		Thr_Cost = inf	135	5.073	138636	11.4
bnrE	Receiver Initiated	round robin	150	1.210	87572	—
		Thr_Cost = 30	145	0.970	76334	12.8
		Thr_Cost = 1000	139	1.217	51582	41.1
		Thr_Cost = inf	140	4.043	39858	54.5
bnrE	Mixed	round robin	146	1.519	245270	—
		Thr_Cost = 30	141	1.411	218568	10.9
		Thr_Cost = 1000	142	1.601	179372	26.9
		Thr_Cost = inf	140	4.864	169992	30.7
MDC	Sender Initiated	round robin	150	2.171	236304	—
		Thr_Cost = 30	149	1.789	231310	2.1
		Thr_Cost = 1000	147	1.909	225912	4.4
		Thr_Cost = inf	148	5.323	223004	5.6
MDC	Receiver Initiated	round robin	156	1.635	85646	—
		Thr_Cost = 30	155	1.203	70236	18.0
		Thr_Cost = 1000	153	1.192	50308	41.3
		Thr_Cost = inf	152	4.479	28132	67.2
MDC	Mixed	round robin	153	2.208	324914	—
		Thr_Cost = 30	158	1.707	291428	10.3
		Thr_Cost = 1000	150	1.850	249578	23.2
		Thr_Cost = inf	150	5.429	236804	27.1

which are not very sensitive to locality, the reduction in network traffic is between the two other cases. The benefit gained by exploiting locality in the mixed sender/receiver case is larger than in sender initiated, and smaller than in receiver initiated.

Of course, locality also has an effect on the execution time of the application. As one improves the locality of the application, fewer update messages are needed, and the time the processors spend sending and receiving messages is reduced. This has a direct effect on the execution time. Unfortunately, there is another opposing effect as well. If all wires are assigned to processors strictly on the basis of locality, it is likely that the resulting wire assignment will not be load balanced. This effect will be discussed in Section 5.3.2.

5.3 Limitations on Exploiting Locality

The previous subsections have shown that exploiting locality to reduce execution time and increase quality clearly has some benefit. Unfortunately, there are several factors which limit the amount to be gained by taking advantage of locality in a problem. First, the standard cell

circuits themselves have only a limited amount of locality. If the wires to be routed are long enough to pass through the owned regions of several processors, there is an unavoidable amount of interprocessor communication that will take place to perform the necessary updates. Further, as the number of processors increases, the region size will decrease, and the locality will decrease as well. Second, fully exploiting the locality that does exist can often interfere with the load balancing of the processors. If there are many wires within a single processor's region, then considering locality alone, that processor should route them all. However, this may cause that processor to have a disproportionate amount of work, resulting in a load imbalance and poor performance. These two issues are treated in the sections below.

5.3.1 Limited Circuit Locality

To determine an upper bound on the degree to which LocusRoute can take advantage of locality, we developed a measure of the amount of locality in the standard cell circuits being used. Using this measure, we found that the degree to which locality can be exploited is, in part, limited by the circuits themselves. This section will first describe the method of measuring circuit locality, and then analyse the results for the two benchmark circuits.

The measure is computed in the following steps. First, wires are assigned to processors using one of the methods already described. Next, for each processor, the program computes the number of wire segments to be routed in each region of the cost array, and the distance in hops from the region where the segment lies to the processor performing the routing. Locality is considered to be the average distance between the routing processor and the processor that owns a region, weighted by the number of wire segments routed at this distance. Thus, a low number means good locality. For example, a locality measure of 0 indicates that all segments were routed by the region owner, giving perfect locality. Increases in the locality measure indicate that the average segment is being routed at a distance further from the owner. Note that when a wire assignment with infinite ThresholdCost is used for this calculation, one end of the wire is guaranteed to be in the owned region of the processor doing the routing. In this case, the degree of locality is mainly a measure of the length of the wire, compared to the size of the individual cost array regions because it measures how many hops the other end of the wire is from the lower left end, which is certainly in the region of the routing processor.

Computing the locality for the two test circuits, using several wire allocation strategies, gave the results shown in Table 5. These results are computed assuming 16 processors. The results indicate that the amount of locality to be exploited in these circuits is limited. For the bnrE circuit, using a round robin assignment, the average segment gets routed by a processor 1.96 hops away from the processor that owns the region the segment lies in. When the wire assignment method is changed to one with ThresholdCost equal to infinity, the average segment is routed by a processor only 1.24 hops away. As the number of processors is increased, the locality in the circuit will decrease because the size of each owned region, formed by splitting the cost array into equal chunks, will decrease. This limited locality in the circuits indicates that the message passing approach may require substantially more message traffic with very large numbers of processors than it currently does, and that the solution quality will be degraded as well.

5.3.2 Tradeoff between Locality and Load Balancing

The requirement that a parallel program be load balanced is also a limitation on the benefit possible from methods which exploit locality. To some extent, a circuit with good locality will

Table 5: Circuit locality.

Circuit	Allocation strategy	Total Wires	Wires Held for Round_robin asmt.	Measure of Locality
bnrE	round robin	420	420	1.96
	ThresholdCost = 30		209	1.77
	ThresholdCost = 1000		25	1.30
	ThresholdCost = inf		0	1.24
MDC	round robin	573	573	2.05
	ThresholdCost = 30		263	1.58
	ThresholdCost = 1000		38	1.04
	ThresholdCost = inf		0	0.99

require fewer updates, and therefore, less time to execute. However, the effect of a load imbalance can outweigh the subtle effect of the difference in update time. Therefore, in terms of execution time, the optimal point is neither a fully load balanced circuit, where the update time becomes significant, nor a fully local circuit, where the load imbalances become significant, but rather a point between the two.

The data from Table 4 shows this quite clearly, with the optimal execution time in almost every case being the ThresholdCost = 30 wire assignment. The most obvious example of the negative effect of locality on load balancing is exhibited in the move from the point with infinite ThresholdCost to the point with ThresholdCost equal to 1000. For example in the MDC circuit, this change in the way the last 38 wires are assigned gives as much as a 73% execution time reduction. However, wire assignments which use the most locality generally give the best solution quality. When processors route in localized regions, each has a fairly consistent view of the area it is routing in. Ultimately, this is a more effective way to produce good solution quality than nonlocalized routing with periodic updates.

6 Conclusions

The goals of this research were to re-evaluate the tradeoffs between shared memory and message passing architectures in light of the new features becoming prevalent in the two architectures. Specifically, we studied the level of traffic required by each approach to maintain consistency, and the effect of exploiting locality on this traffic. Although this study provides data from a single application, we feel that it is representative of a class of applications which do not require the strict consistency enforced by hardware cache coherence schemes.

We show that implementing the LocusRoute application on a message passing machine can result in a dramatic decrease in the amount of interconnection network traffic, with only a small negative effect on the solution quality. This is especially impressive because LocusRoute has been touted as an excellent application for a shared memory architecture. However, this dramatic improvement did have a cost. The explicit control afforded, and in fact required, by the message passing architecture requires significantly larger programming effort. The decisions of how to partition the cost array among the processors, how to initiate updates, how frequently updates should occur, and how to assign wires to processors, all involve complex tradeoffs and

much programming.

We further show that exploiting locality in the message passing case can have a positive effect on all three of the factors studied in this paper: solution quality, network traffic, and to a lesser extent, execution time. What, then, is the cost of exploiting it? The answer, once again, is that exploiting locality currently requires more programmer effort than using a simpler method which does not exploit it.

In this paper, we also studied the shared memory approach, examining the traffic necessary to maintain cache consistency. We found that the bus traffic in a shared memory architecture can be as much as two orders of magnitude larger than the network traffic in a shared memory approach. In an absolute sense, however, the amount of traffic is not excessive. For this reason, in applications where the improved solution quality given by the shared memory implementation is important, it appears to be the correct choice. Perhaps the most compelling benefit, however, of the shared memory architecture is the easy and natural programming environment it provides. The hardware cache coherence enforced enables programs to be developed and debugged more quickly and easily.

We also presented data indicating that traditional bus-based shared memory machines are not extremely sensitive to exploiting locality. In the LocusRoute application, this is in part due to the difficulty of singling out the array elements needed by each processor. In general, even in the most local wire assignment method, interference between processors is still a problem. Sharing of cost array information leads to a large number of invalidations and refetches. However, future machines relying on hierarchies to scale the total number of processors, are expected to be more sensitive to locality. As these architectures become available, more research will be needed to automatically detect and exploit locality in parallel programs.

7 Acknowledgements

We would like to thank Jonathan Rose for his patience in explaining the LocusRoute application to us, and Andreas Nowatzyk for his prompt, helpful replies to questions about CBS. Margaret Martonosi is supported by a fellowship from the National Science Foundation. Anoop Gupta is supported by DARPA contract N00014-87-K-0828 and by a faculty award from Digital Equipment Corporation.

References

[1] Anant Agarwal. Richard Simoni. John Hennessy. and Mark Horowitz.
Scalable Directory Schemes for Cache Coherence.
In *Proc. 15th Annual International Symposium on Computer Architecture*. June 1988.

[2] James Archibald and Jean-Loup Baer.
Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model.
ACM Transactions on Computer Systems. 4(4):273–298, November 1986.

[3] William C. Athas and Charles L. Seitz.
Multicomputers: Message–Passing Concurrent Computers.
IEEE Computer, 21(8):9–24, August 1988.

[4] David Cheriton. Anoop Gupta. Patrick Boyle. and Hendrik Goosen.
The VMP Multiprocessor: Initial Experience, Refinements. and Performance Evaluation.
In *Proc. Fifteenth Annual Symposium on Computer Architecture*. 1988.

[5] Encore Computer Corp.
Multimax Technical Summary.
1986.

[6] W. J. Dally and C. L. Seitz.
Deadlock-Free Message Routing in Multiprocessor Interconnection Networks.
IEEE Trans. Computers. 36(5):547–553. May 1987.

[7] William J. Dally.
A VLSI Architecture for Concurrent Data Structures.
Kluwer Publishers, 1987.

[8] William J. Dally.
Wire Efficient VLSI Multiprocessor Communication Networks.
In *Stanford Conference on Advanced Research in VLSI*. pages 391–415, 1987.

[9] William J. Dally. Linda Chao. et al.
Architecture of a Message-Driven Processor.
In *Proc. 14th Annual International Symposium on Computer Architecture*. June 1987.

[10] Ametek Computer Research Division.
Series 2010 System General Description Issue 3.
1988.

[11] J.P. Hayes et al.
A Microprocessor-based Hypercube Supercomputer.
IEEE Micro. 6(5):6–17. October 1986.

[12] BBN Laboratories Inc.
Butterfly Parallel Processor Overview.
1986.
BBN Report No. 6148.

[13] Andrew W. Wilson Jr.
Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors.
In *Proc. 14th Annual International Symposium on Computer Architecture*, pages 244–251.
June 1987.

[14] C. R. Lang Jr.
The Extension of Object–Oriented Languages to a Homogeneous. Concurrent Architecture.